# Compound Interest - Dealing with Two Decades of Technical Debt in Embedded Linux

Bartosz Golaszewski

Embedded Open Source Summit 2024, Seattle, WA

Linaro

# About

- Linux kernel developer for the Qualcomm Landing Team at Linaro
- 15 years of embedded linux experience
- Maintainer of the GPIO subsystem
- Author and maintainer of libgpiod
- Open-source contributor to many other projects
- Interested in complex software architecture

# Linaro is the software engine of the Arm Ecosystem

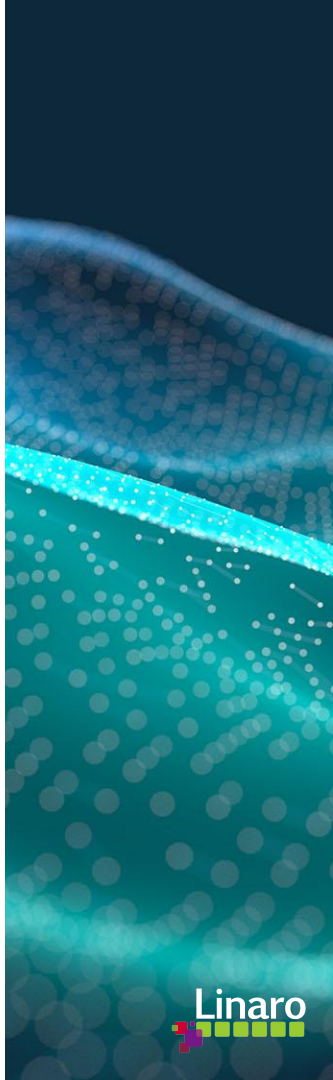
## Linaro empowers rapid product deployment within the dynamic Arm Ecosystem.

- Our cutting-edge solutions, services and collaborative platforms facilitate the swift development, testing, and delivery of Arm-based innovations, enabling businesses to stay ahead in today's competitive technology landscape.
- Our expertise and contributions spread from Testing & LTS, Security, Cloud & Edge Computing, IoT, AI, CI/CD, Toolchain and Virtualization to vertical projects like Windows on Arm and Android Ecosystem enabling and maintenance.
- Linaro fosters and environment of collaboration, standardization and optimization among businesses and open source ecosystems to accelerate the deployment of Arm-based products and technologies along with representing a pivotal role in open source discovery and adoption.

**Linaro has enabled trust, quality and collaboration since 2010**

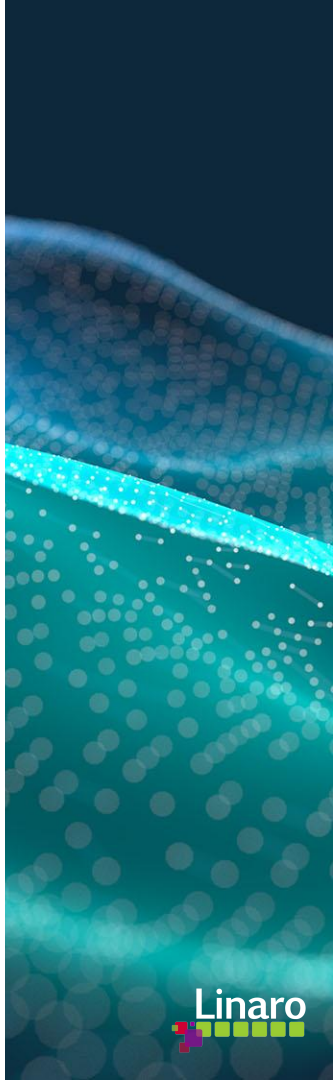
Linaro

# What is “technical debt” in software?



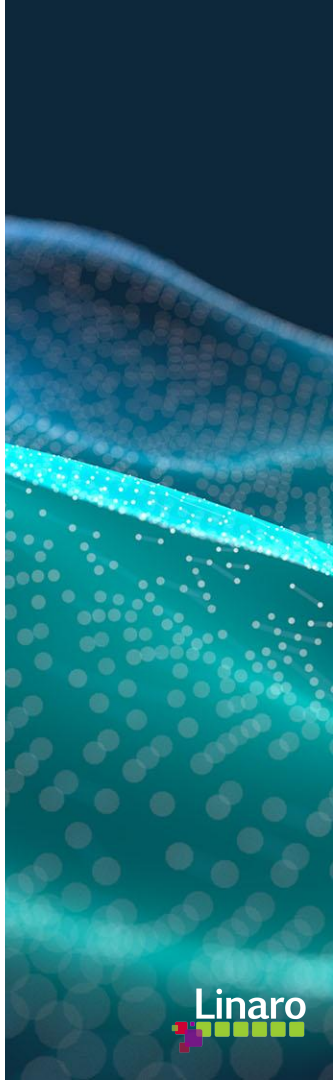
# What is “technical debt” in software?

- Borrowing time now
  - Use a sub-optimal solution
  - Ship faster



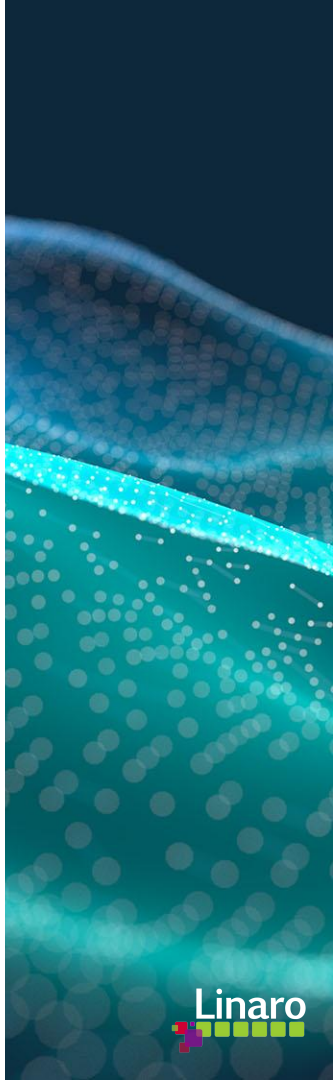
# What is “technical debt” in software?

- Borrowing time now
  - Use a sub-optimal solution
  - Ship faster
- Paying it back later (with interest)
  - Do more work later to fix the issues introduced by the “quick” solution AND implement the “correct” solution



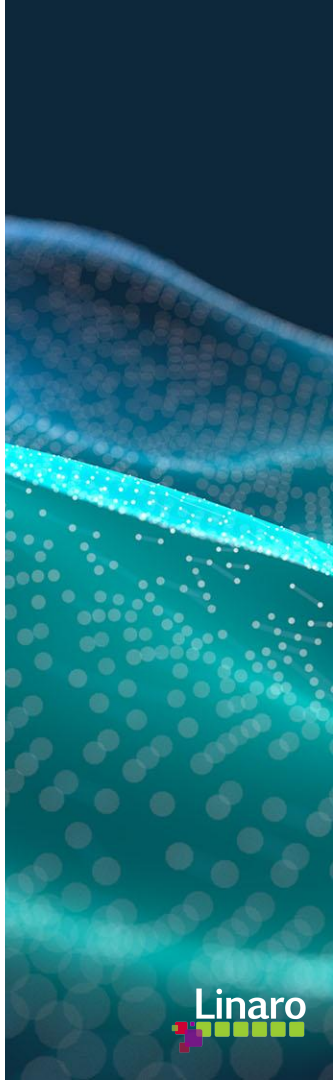
# What is “technical debt” in software?

- Borrowing time now
  - Use a sub-optimal solution
  - Ship faster
- Paying it back later (with interest)
  - Do more work later to fix the issues introduced by the “quick” solution AND implement the “correct” solution
- Debt can be rolled-over
  - Stack workarounds and accrue more technical debt



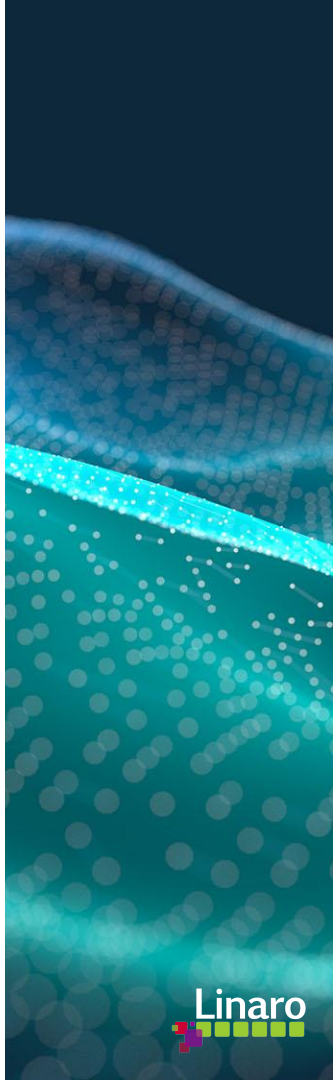
# What is “technical debt” in software?

- Borrowing time now
  - Use a sub-optimal solution
  - Ship faster
- Paying it back later (with interest)
  - Do more work later to fix the issues introduced by the “quick” solution AND implement the “correct” solution
- Debt can be rolled-over
  - Stack workarounds and accrue more technical debt
  - Interest compounds
- Just like real debt: gets hard to keep up with after a certain point



# What is “technical debt” in software?

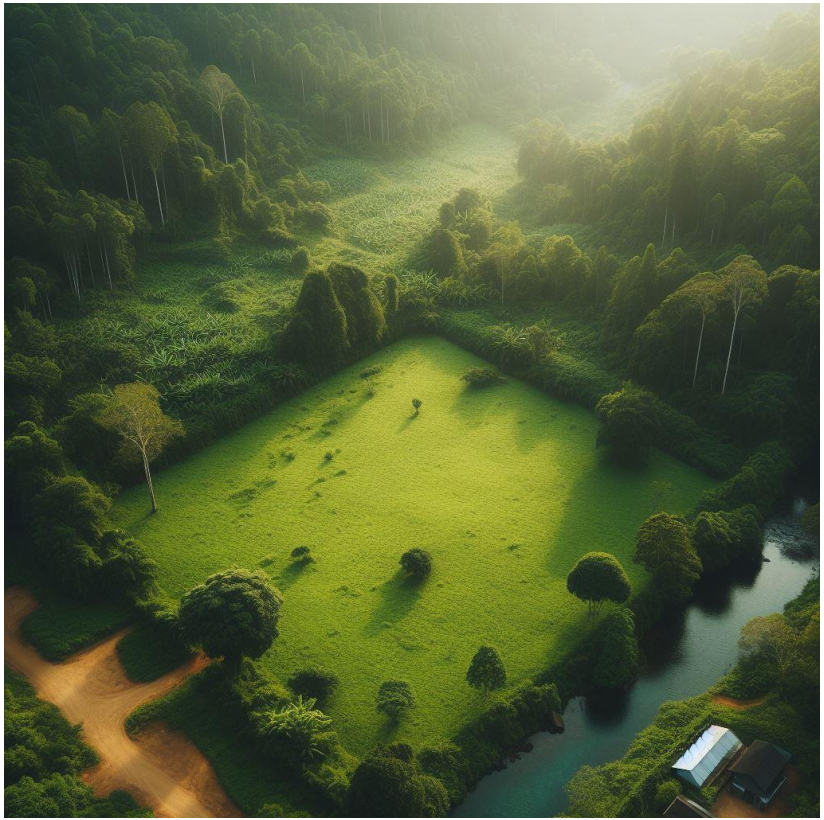
- Borrowing time now
  - Use a sub-optimal solution
  - Ship faster
- Paying it back later (with interest)
  - Do more work later to fix the issues introduced by the “quick” solution AND implement the “correct” solution
- Debt can be rolled-over
  - Stack workarounds and accrue more technical debt
  - Interest compounds
- Just like real debt: gets hard to keep up with after a certain point
- Worst kind of technical debt in programming is not the implementation details, it’s data structures and most importantly: programming interfaces!



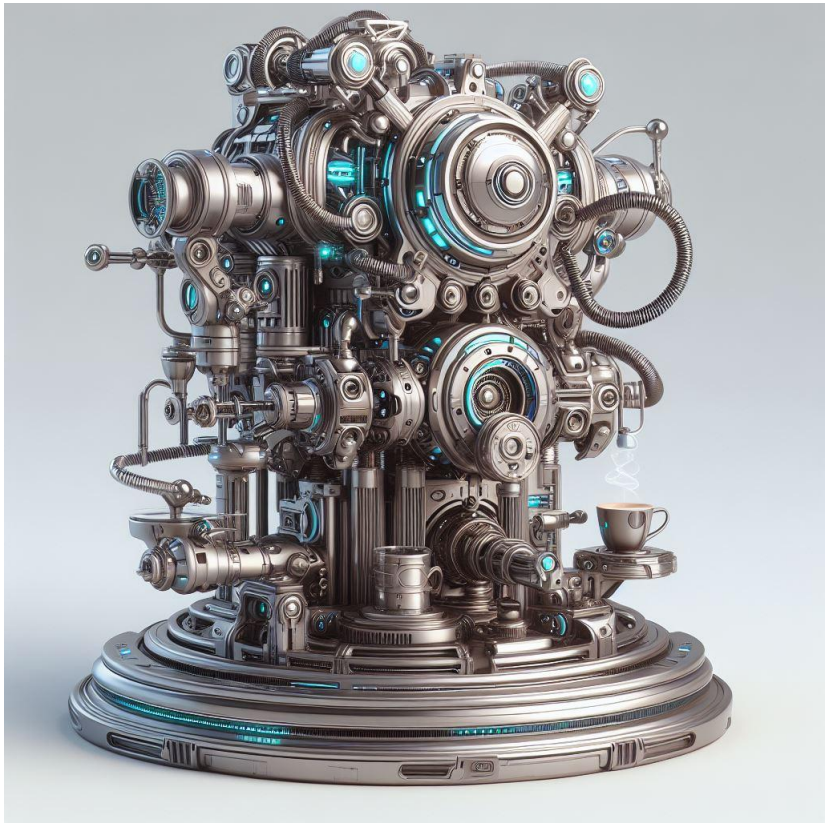
# So you want to start working on a piece of software?

Linaro

# Case 1: Greenfield development



# Case 2: Well-oiled machinery

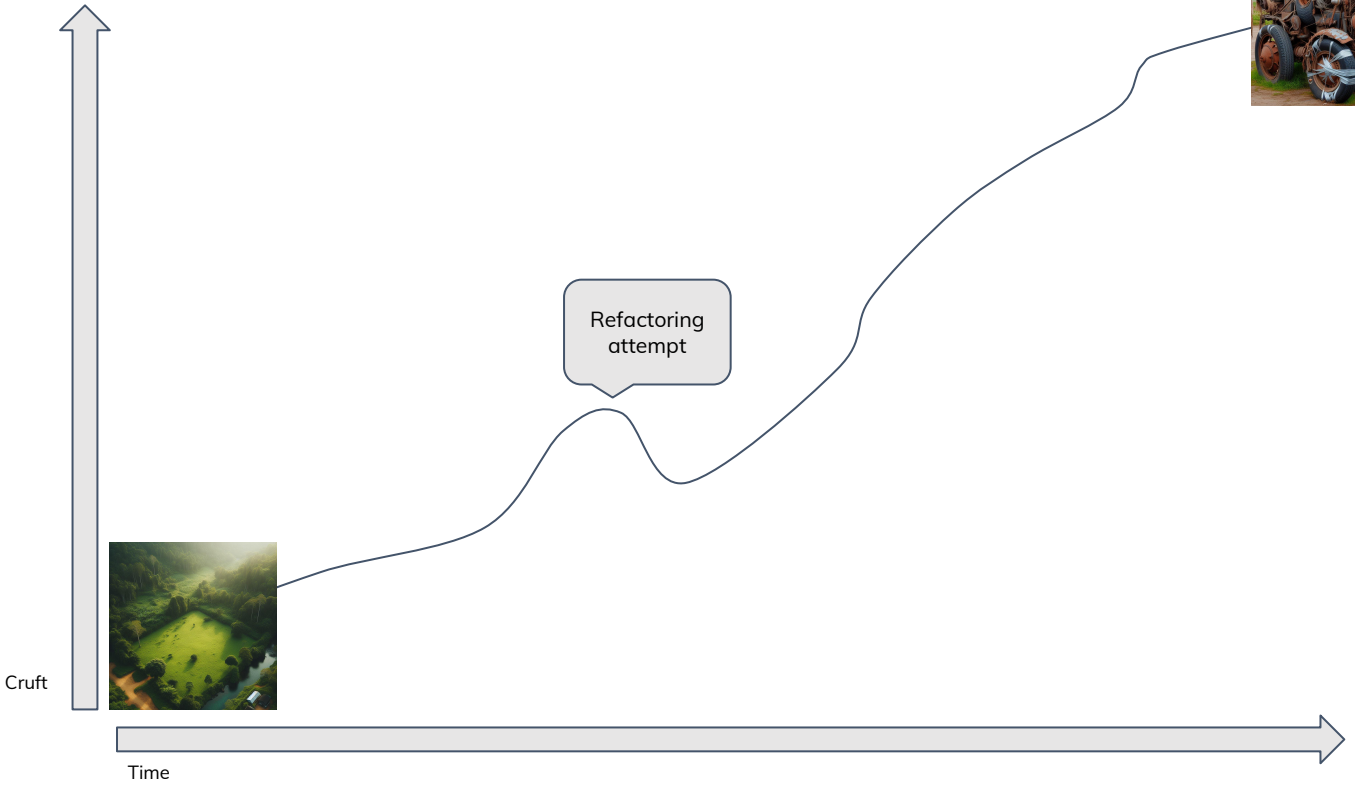


# Case 3: Ducty McTapeface

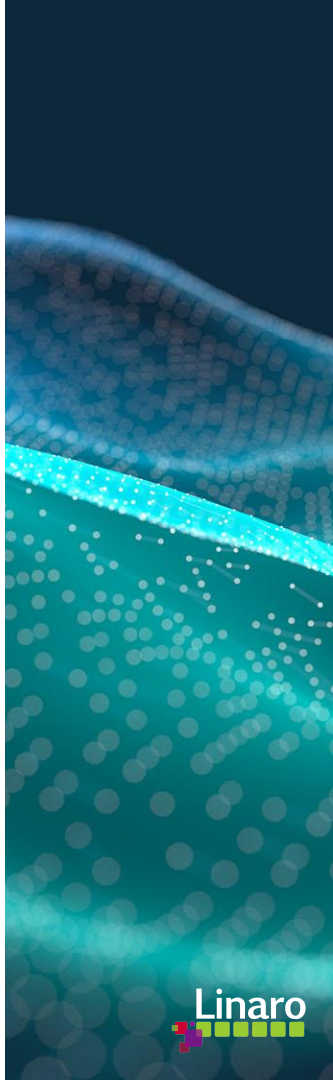
# Case 3: Ducty McTapeface

It works but at what cost...

# How does it happen (generally)?

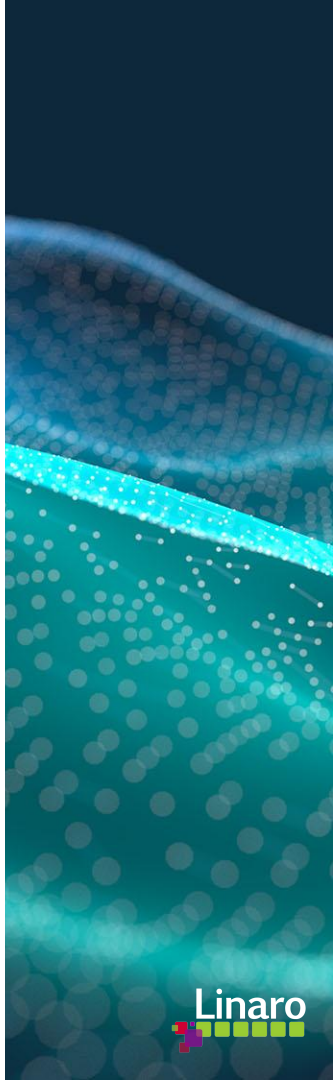


# How does it happen (generally)?



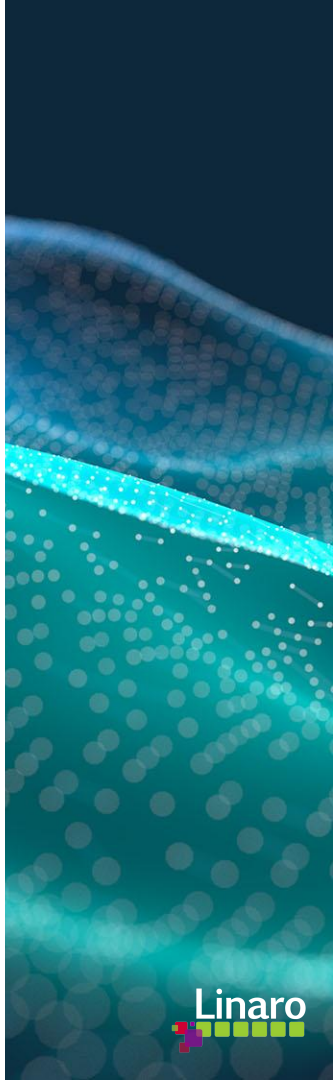
# How does it happen (generally)?

- Uncertain scope & feature creep
  - Use-cases two years from now may be different
  - Users request more and more new features (often valid ones!)
  - Sometimes one just doesn't know what they don't know



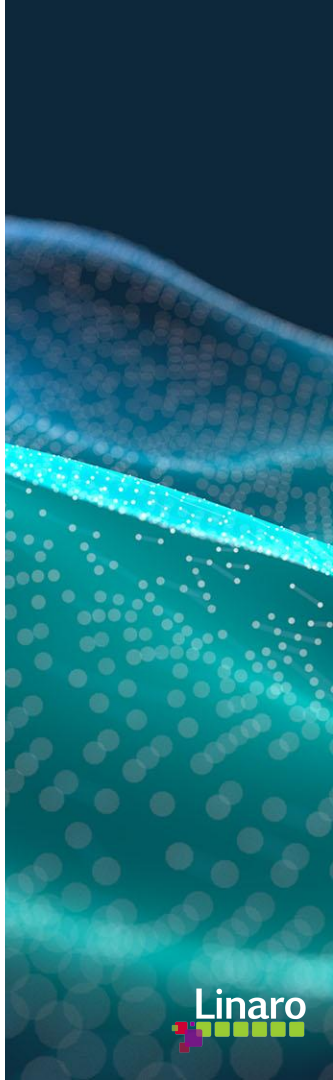
# How does it happen (generally)?

- Uncertain scope & feature creep
  - Use-cases two years from now may be different
  - Users request more and more new features (often valid ones!)
  - Sometimes one just doesn't know what they don't know
- New requirements vs API/ABI stability
  - Old use-cases need to be supported
  - Code using previous minor versions must still build
  - Even the kernel has some sort of in-kernel API stability: modifying an interface used in 100s of places is complex
    - Not to mention the user-space ABI stability

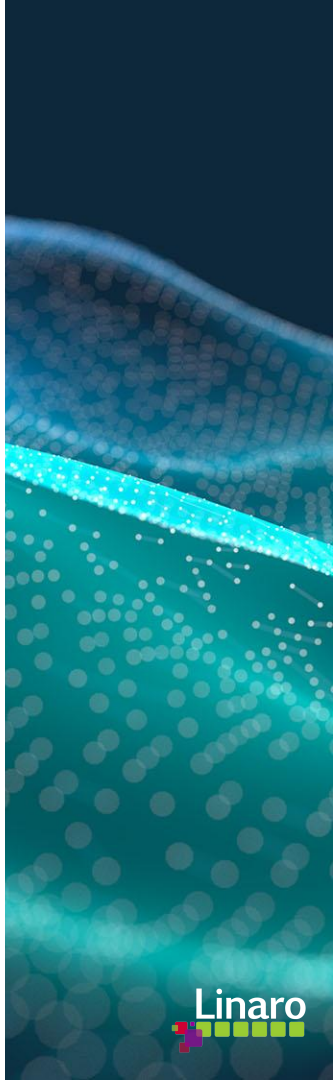


# How does it happen (generally)?

- Uncertain scope & feature creep
  - Use-cases two years from now may be different
  - Users request more and more new features (often valid ones!)
  - Sometimes one just doesn't know what they don't know
- New requirements vs API/ABI stability
  - Old use-cases need to be supported
  - Code using previous minor versions must still build
  - Even the kernel has some sort of in-kernel API stability: modifying an interface used in 100s of places is complex
    - Not to mention the user-space ABI stability
- Lax maintenance
  - Kernel maintainers are often grumpy for a reason
    - Best way to deal with technical debt is to never allow it in the first place
  - Business priorities take over

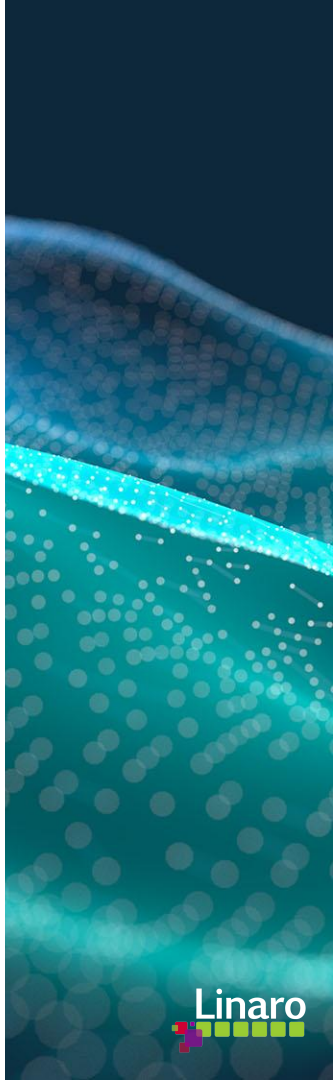


# How does it happen in the kernel?



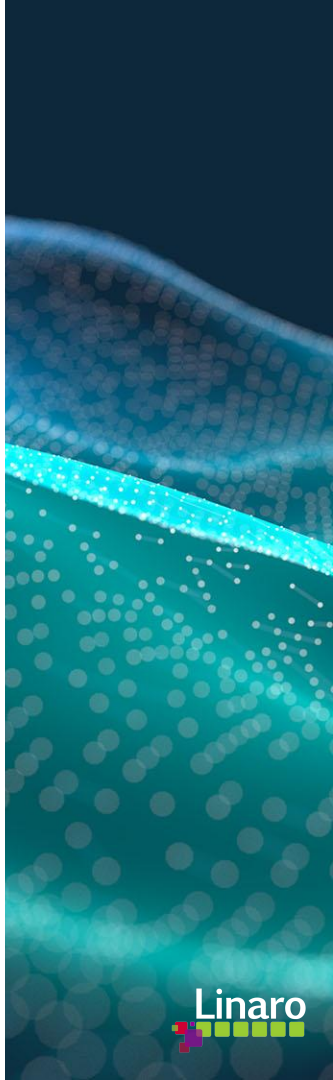
# How does it happen in the kernel?

- Some API and ABI stability is always required
  - If you want to change an API, you need to update the users too
  - If the users are many, it gets tricky
    - Many maintainer trees to target
    - Add new API, convert users one by one, remove old API
    - Easily takes years
  - User-space ABI must not change



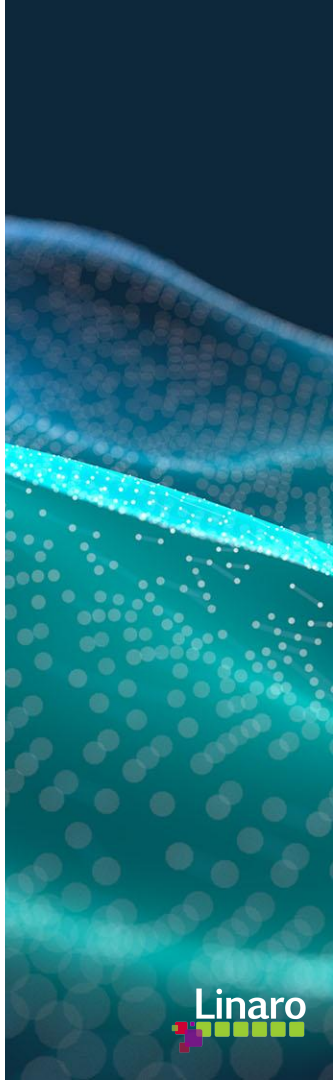
# How does it happen in the kernel?

- Some API and ABI stability is always required
  - If you want to change an API, you need to update the users too
  - If the users are many, it gets tricky
    - Many maintainer trees to target
    - Add new API, convert users one by one, remove old API
    - Easily takes years
  - User-space ABI must not change
- Lack of consistency/redesigning the wheel resulting from a very large codebase
  - Every subsystem does their own thing
  - Copy/paste of a sub-optimal solution from one place that doesn't match the needs of another
  - Not reusing an existing good solution that should be reused

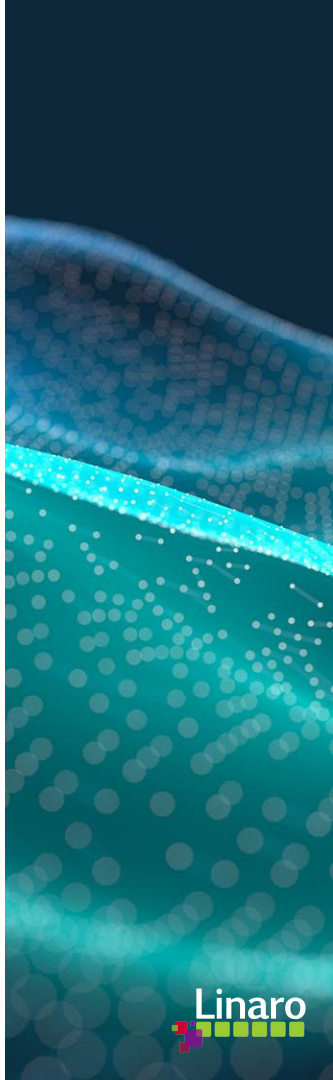


# How does it happen in the kernel?

- Some API and ABI stability is always required
  - If you want to change an API, you need to update the users too
  - If the users are many, it gets tricky
    - Many maintainer trees to target
    - Add new API, convert users one by one, remove old API
    - Easily takes years
  - User-space ABI must not change
- Lack of consistency/redesigning the wheel resulting from a very large codebase
  - Every subsystem does their own thing
  - Copy/paste of a sub-optimal solution from one place that doesn't match the needs of another
  - Not reusing an existing good solution that should be reused
- Lack of a good deprecation mechanism
  - Not possible to always catch people using interfaces documented as deprecated without enforcing it through warnings

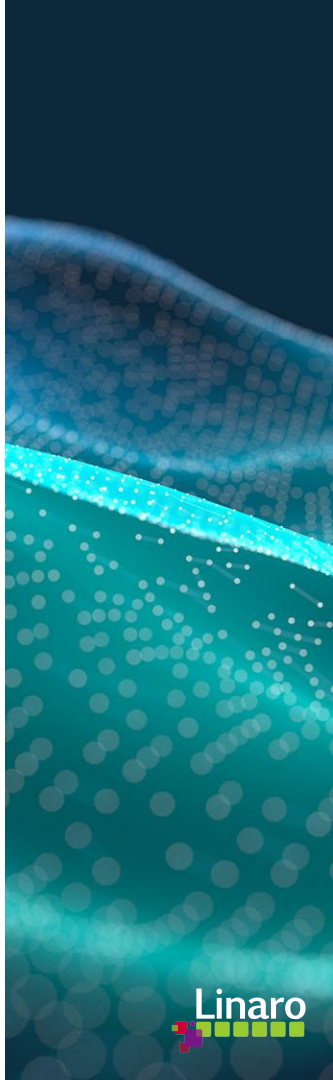


# Practical example: The GPIO subsystem



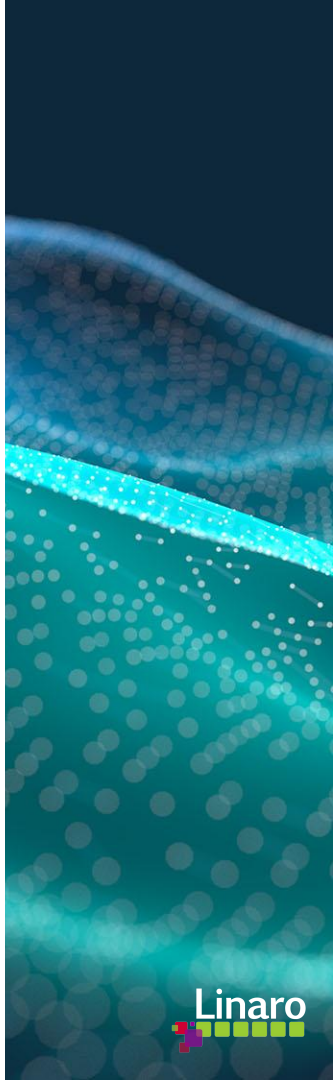
# Practical example: The GPIO subsystem

- Started out as an ad-hoc set of quasi standardized function prototypes implemented by different architectures



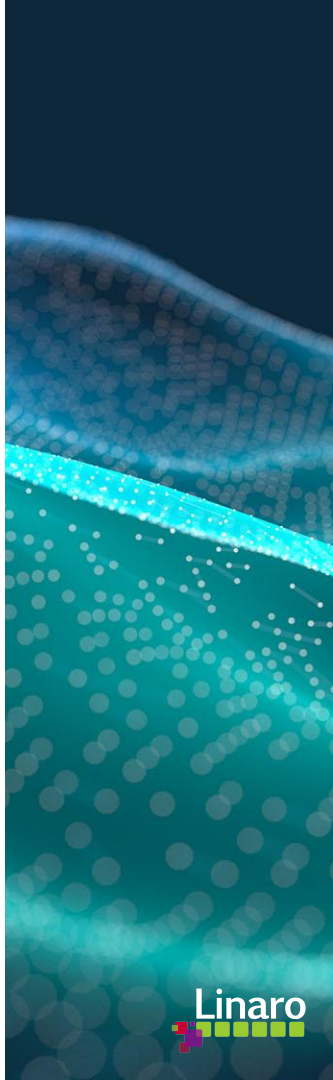
# Practical example: The GPIO subsystem

- Started out as an ad-hoc set of quasi standardized function prototypes implemented by different architectures
- Evolved into a global numberspace-based provider-consumer library



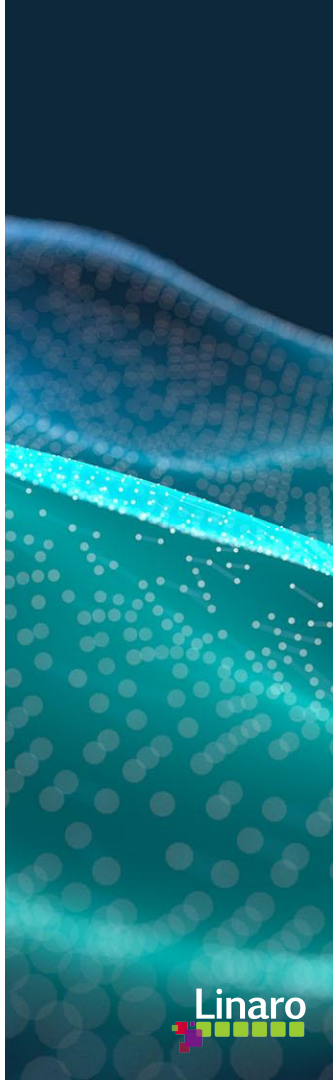
# Practical example: The GPIO subsystem

- Started out as an ad-hoc set of quasi standardized function prototypes implemented by different architectures
- Evolved into a global numberspace-based provider-consumer library
- Eventually became a modern provider-consumer framework based on descriptor handles



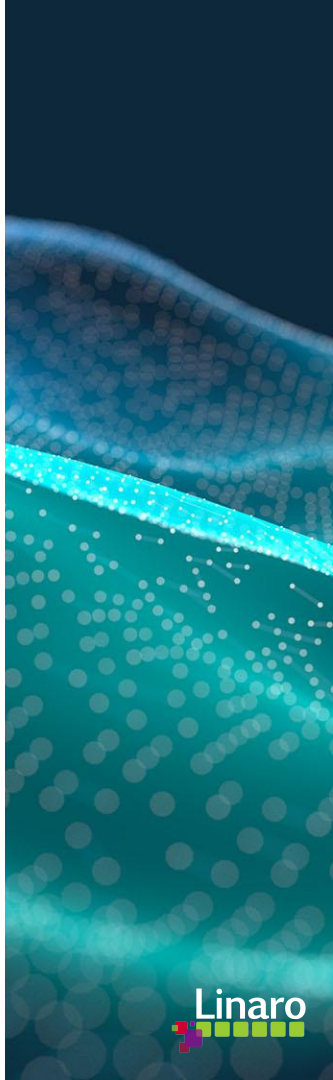
# Practical example: The GPIO subsystem

- Started out as an ad-hoc set of quasi standardized function prototypes implemented by different architectures
- Evolved into a global numberspace-based provider-consumer library
- Eventually became a modern provider-consumer framework based on descriptor handles
  - Still uses the global numberspace internally
  - Old interface is still supported



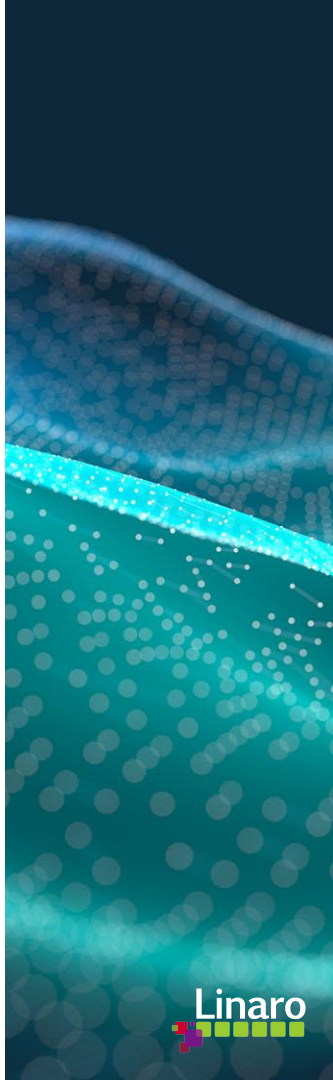
# Practical example: The GPIO subsystem

- Started out as an ad-hoc set of quasi standardized function prototypes implemented by different architectures
- Evolved into a global numberspace-based provider-consumer library
- Eventually became a modern provider-consumer framework based on descriptor handles
  - Still uses the global numberspace internally
  - Old interface is still supported
- Has had multiple maintainers and periods with none at all



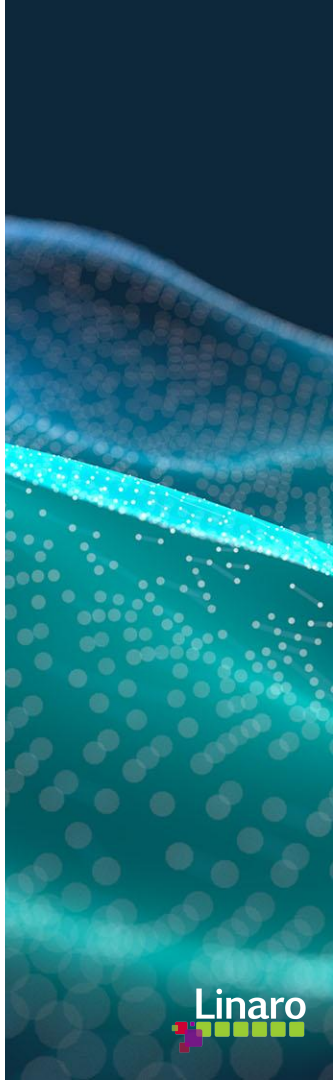
# Practical example: The GPIO subsystem

- Started out as an ad-hoc set of quasi standardized function prototypes implemented by different architectures
- Evolved into a global numberspace-based provider-consumer library
- Eventually became a modern provider-consumer framework based on descriptor handles
  - Still uses the global numberspace internally
  - Old interface is still supported
- Has had multiple maintainers and periods with none at all
- Is ubiquitous in the kernel - both among consumer as well as providers



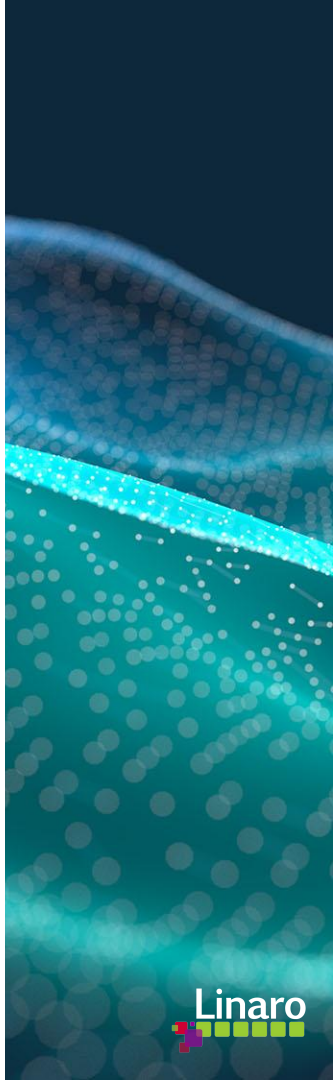
# Practical example: The GPIO subsystem

- Started out as an ad-hoc set of quasi standardized function prototypes implemented by different architectures
- Evolved into a global numberspace-based provider-consumer library
- Eventually became a modern provider-consumer framework based on descriptor handles
  - Still uses the global numberspace internally
  - Old interface is still supported
- Has had multiple maintainers and periods with none at all
- Is ubiquitous in the kernel - both among consumer as well as providers
  - Everyone uses GPIOs
    - Even indirectly: regulators, resets etc. may hide GPIO logic
  - Many devices export GPIOs

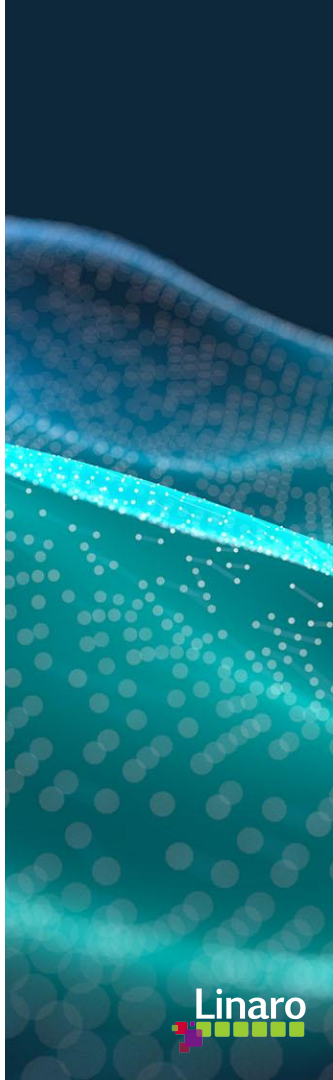


# Practical example: The GPIO subsystem

- Started out as an ad-hoc set of quasi standardized function prototypes implemented by different architectures
- Evolved into a global numberspace-based provider-consumer library
- Eventually became a modern provider-consumer framework based on descriptor handles
  - Still uses the global numberspace internally
  - Old interface is still supported
- Has had multiple maintainers and periods with none at all
- Is ubiquitous in the kernel - both among consumer as well as providers
  - Everyone uses GPIOs
    - Even indirectly: regulators, resets etc. may hide GPIO logic
  - Many devices export GPIOs
- Over the years accrued a significant amount of technical debt

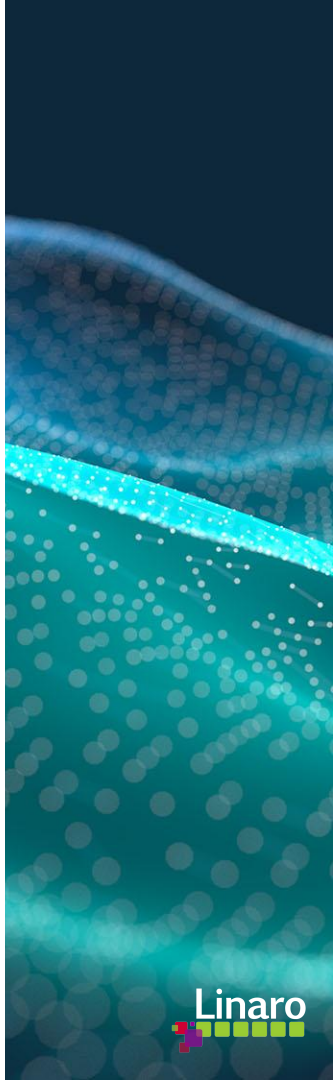


# GPIO prehistory



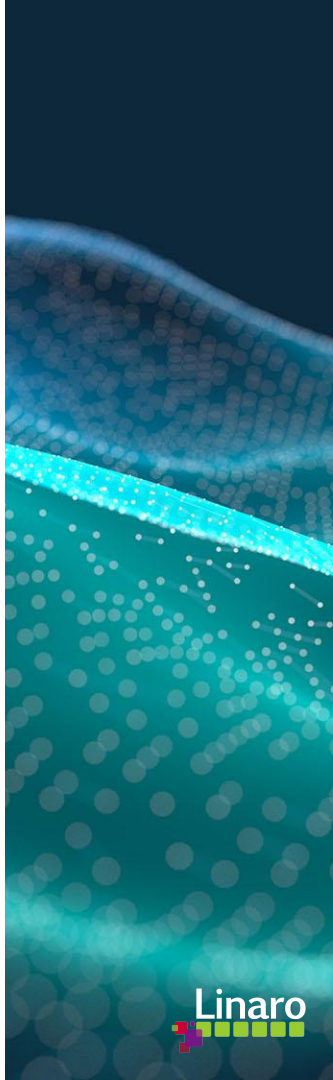
# GPIO prehistory

- Before any unified GPIO subsystem
- Every architecture would do something different
- Set of semi-standardized interfaces existed but not all architectures would implement them



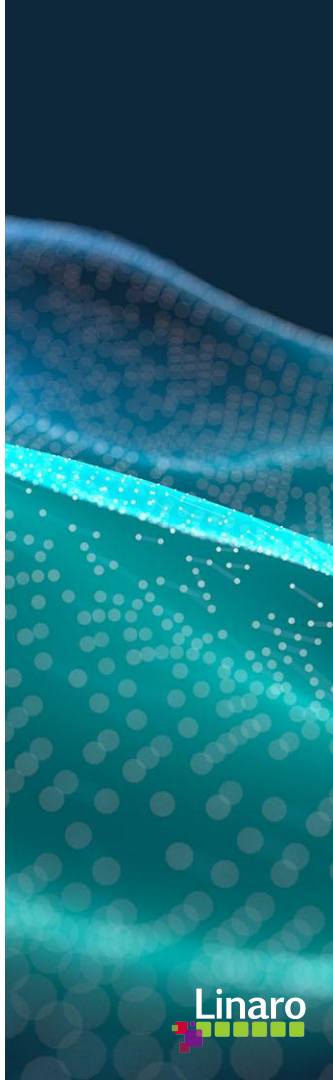
# GPIO prehistory

- Before any unified GPIO subsystem
- Every architecture would do something different
- Set of semi-standardized interfaces existed but not all architectures would implement them
- GPIO “drivers” would live in `arch/`
- Fallout from this still exists in the kernel almost 20 years later!

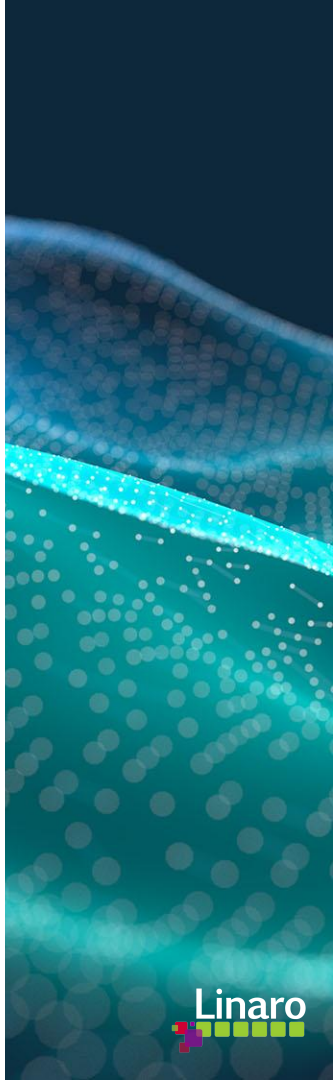


# GPIO prehistory

- Before any unified GPIO subsystem
- Every architecture would do something different
- Set of semi-standardized interfaces existed but not all architectures would implement them
- GPIO “drivers” would live in `arch/`
- Fallout from this still exists in the kernel almost 20 years later!
  - We still have GPIO drivers in `arch/` that don’t use the driver model and are only barely duct taped to the GPIO framework

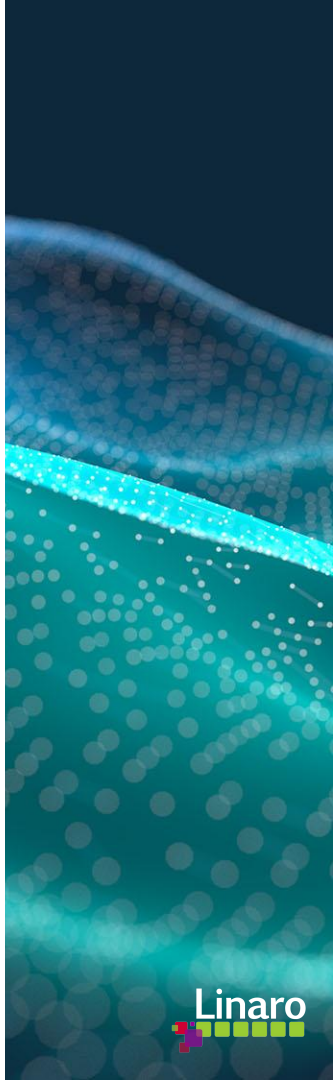


# GPIO global numberspace



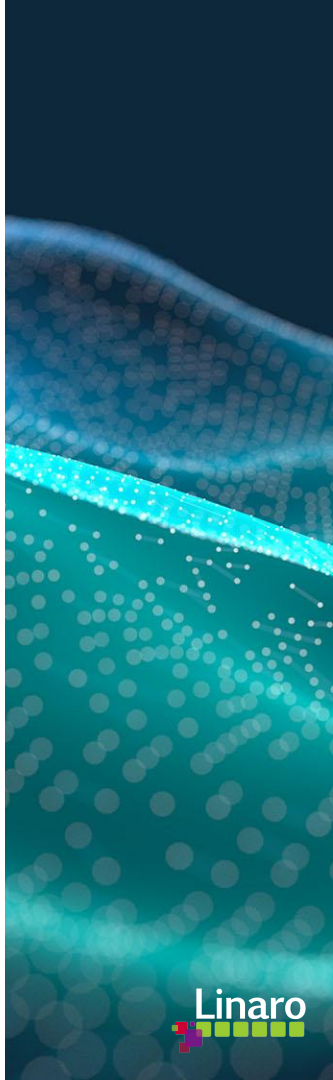
# GPIO global numberspace

- The first provider-consumer model (introduced in 2008) would operate on global GPIO numbers
  - Half hard-coded/half dynamic GPIO numbering



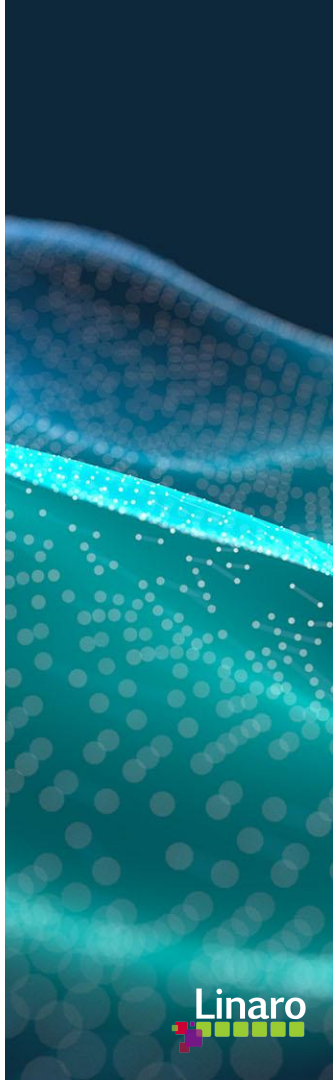
# GPIO global numberspace

- The first provider-consumer model (introduced in 2008) would operate on global GPIO numbers
  - Half hard-coded/half dynamic GPIO numbering
- Worst side-effect of using the numberspace and exporting it to user-space: it’s become expected that GPIO number visible to user-space remain stable

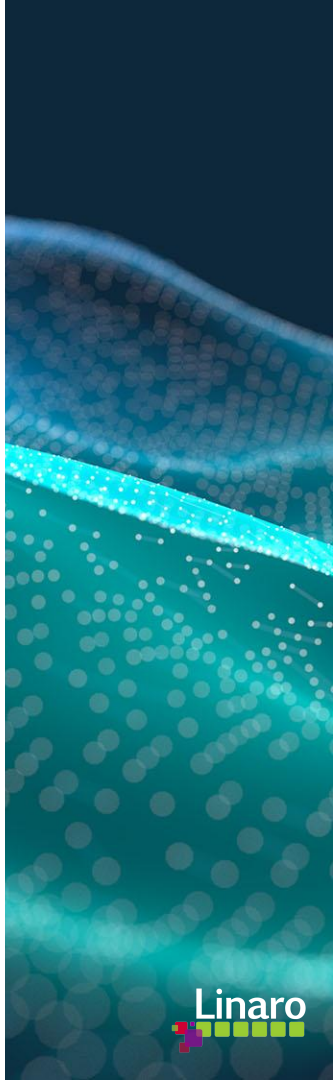


# GPIO global numberspace

- The first provider-consumer model (introduced in 2008) would operate on global GPIO numbers
  - Half hard-coded/half dynamic GPIO numbering
- Worst side-effect of using the numberspace and exporting it to user-space: it's become expected that GPIO number visible to user-space remain stable
- Consumers get numbers assigned, request them and use them
  - Nothing stops a rogue driver from requesting an arbitrarily hard-coded GPIO number

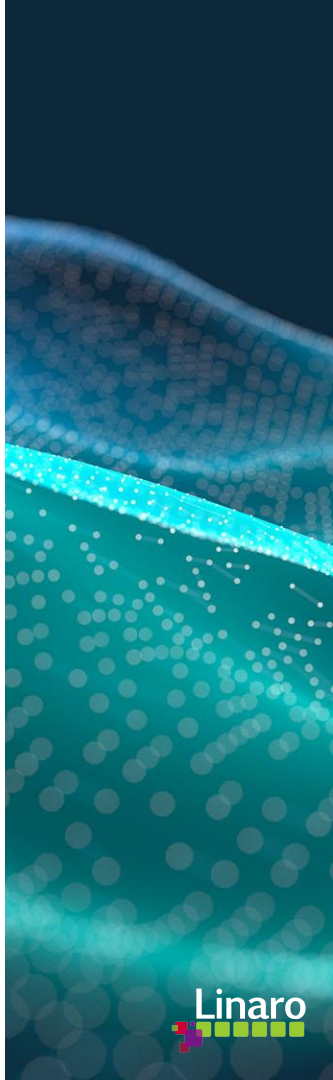


# GPIO descriptors



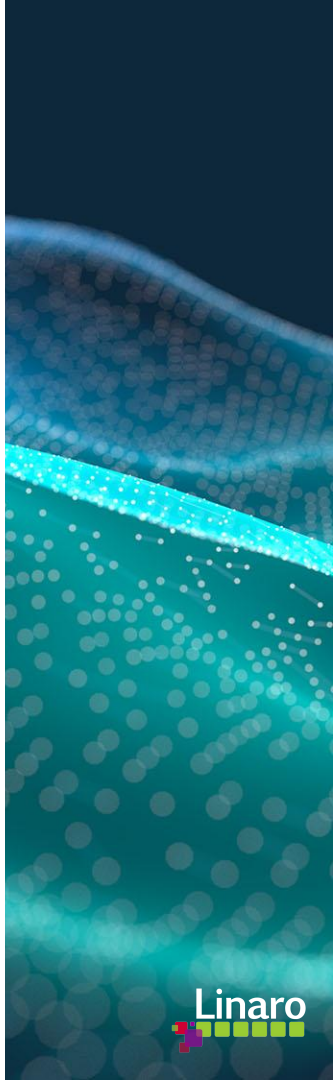
# GPIO descriptors

- Provider drivers would expose GPIO chips
- Platform data/device-tree/ACPI would assign GPIOs to consumers



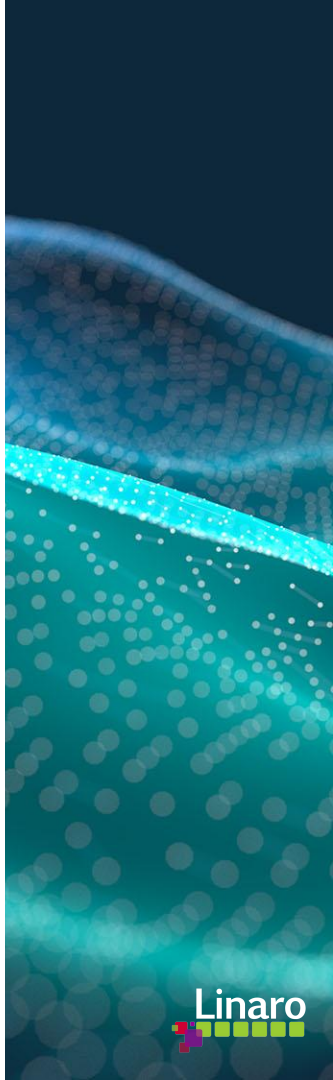
# GPIO descriptors

- Provider drivers would expose GPIO chips
- Platform data/device-tree/ACPI would assign GPIOs to consumers
- Consumer drivers would get opaque handles (descriptors) to GPIOs
  - They cannot get access to GPIOs not assigned to them

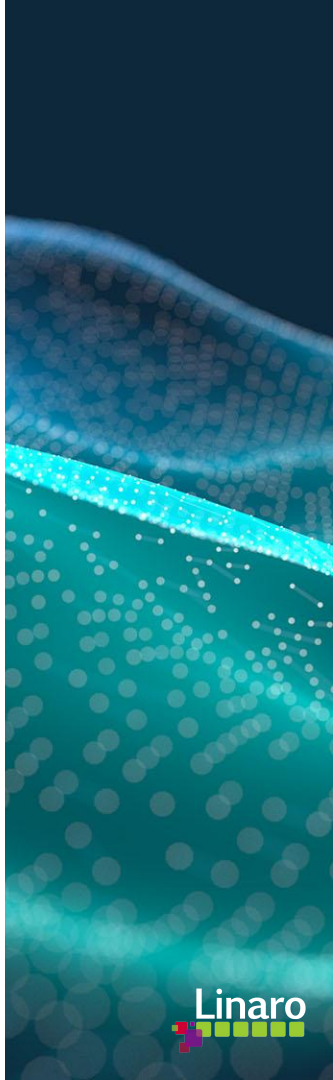


# GPIO descriptors

- Provider drivers would expose GPIO chips
- Platform data/device-tree/ACPI would assign GPIOs to consumers
- Consumer drivers would get opaque handles (descriptors) to GPIOs
  - They cannot get access to GPIOs not assigned to them
- Technical debt strikes back:
  - Internally GPIOLIB still uses the global numberspace and has a whole glue layer between the two interfaces

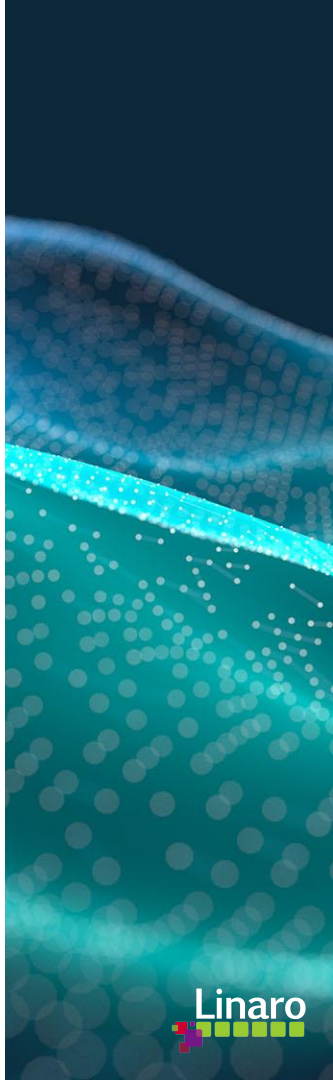


# GPIO numberspace removal



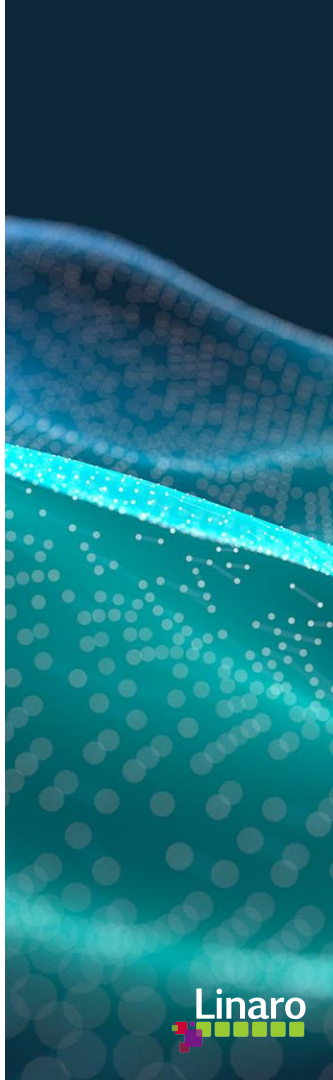
# GPIO numberspace removal

- Work has been ongoing for years
  - Converting users one at a time



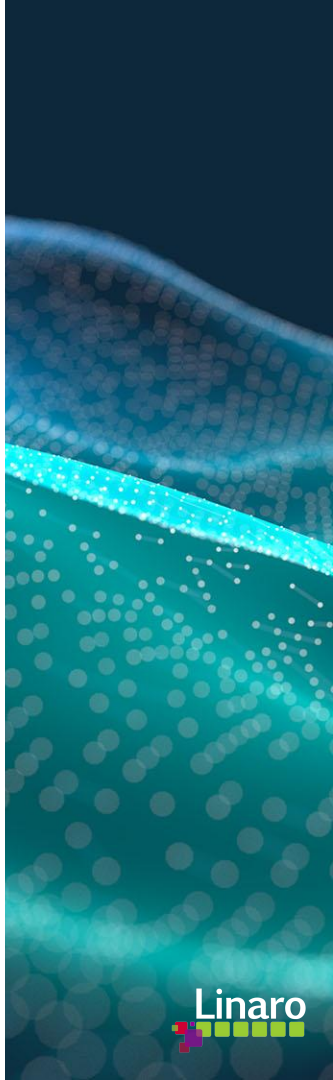
# GPIO numberspace removal

- Work has been ongoing for years
  - Converting users one at a time
- We cannot remove the old interface until we convert all users to the new one

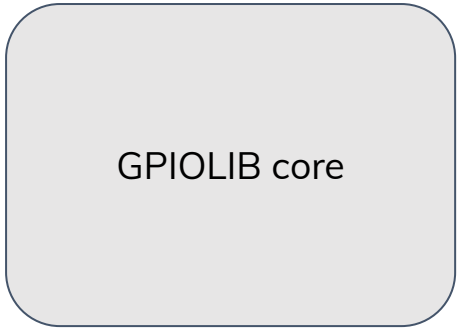


# GPIO numberspace removal

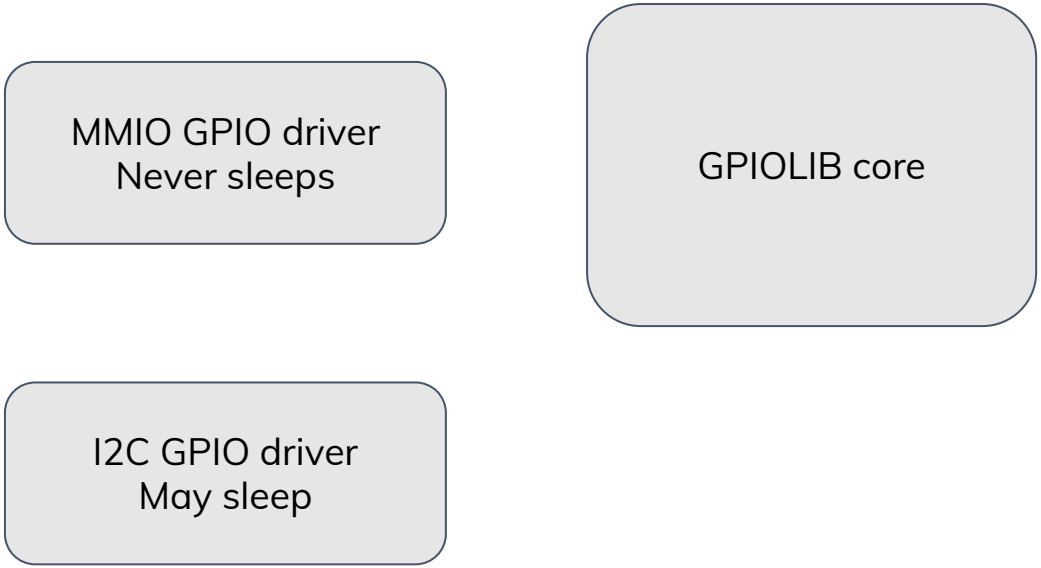
- Work has been ongoing for years
  - Converting users one at a time
- We cannot remove the old interface until we convert all users to the new one
- Even once we convert all in-kernel users, we'll still be left with user-space programs relying on the numbers exported over sysfs
  - Unlike kernel drivers, we cannot enforce the migration of user-space to the descriptor bases uAPI



# Serialization in GPIOLIB



# Serialization in GPIOLIB



# Serialization in GPIOLIB

MMIO GPIO driver
Never sleeps

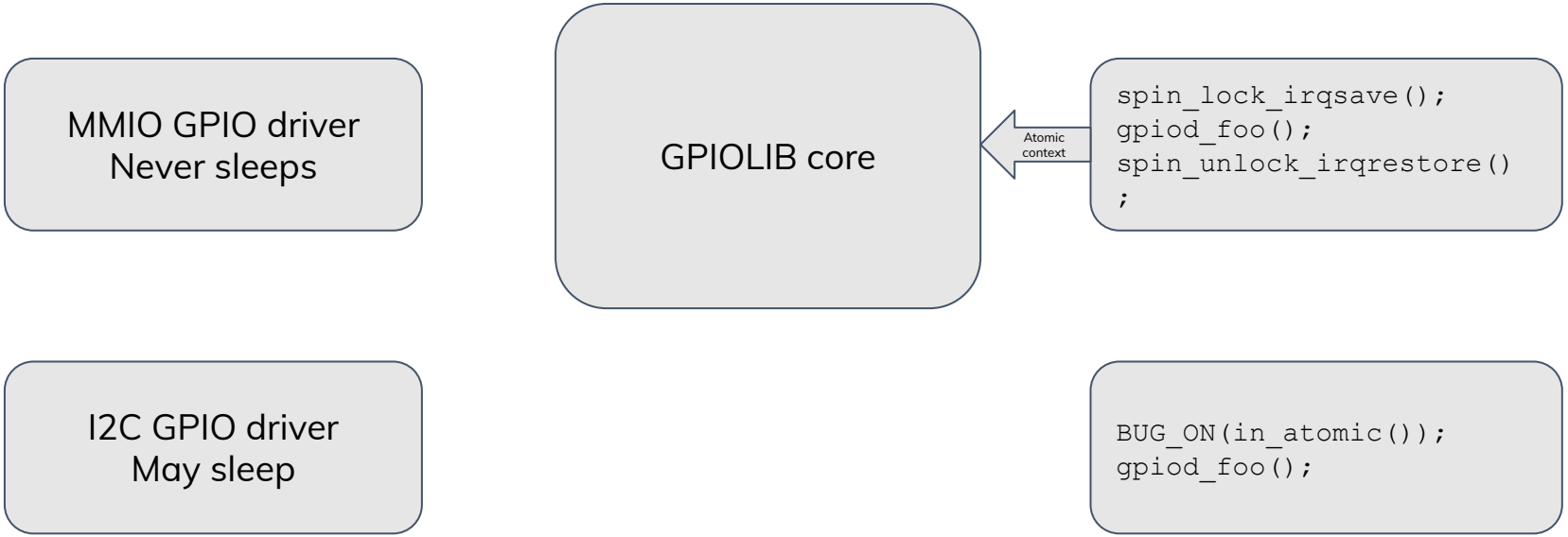
GPIOLIB core

```
spin_lock_irqsave();
gpiod_foo();
spin_unlock_irqrestore()
;
```

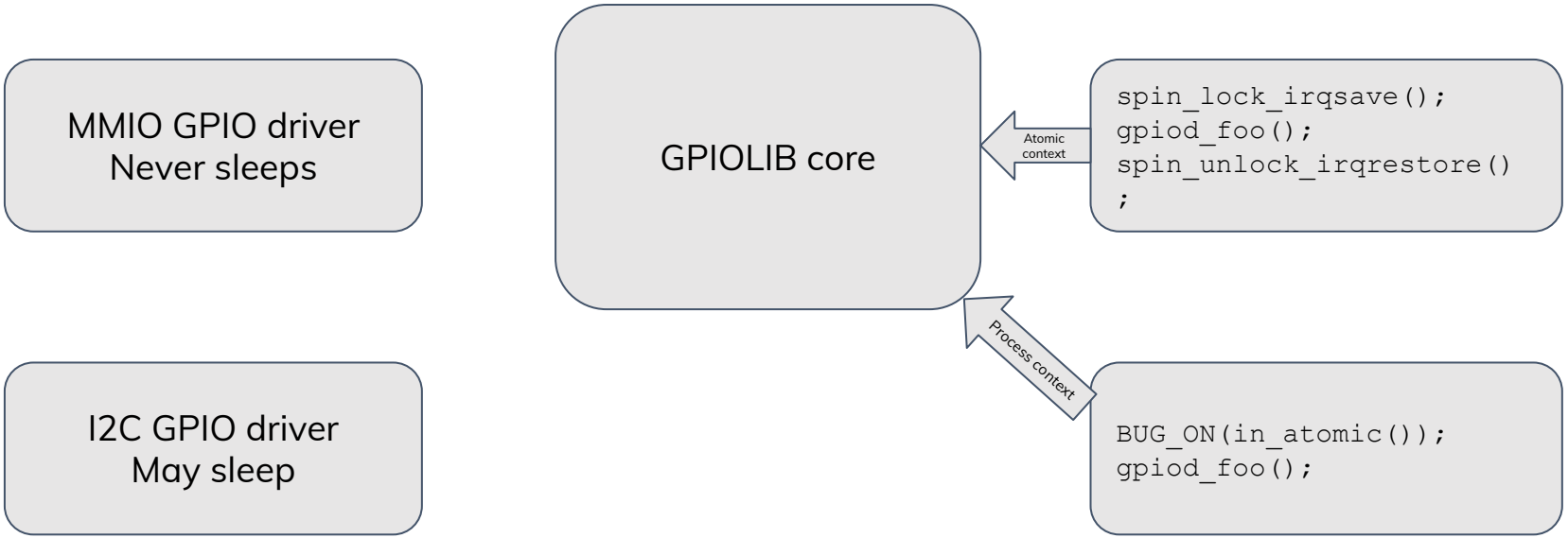
I2C GPIO driver
May sleep

```
BUG_ON(in_atomic());
gpiod_foo();
```

# Serialization in GPIOLIB

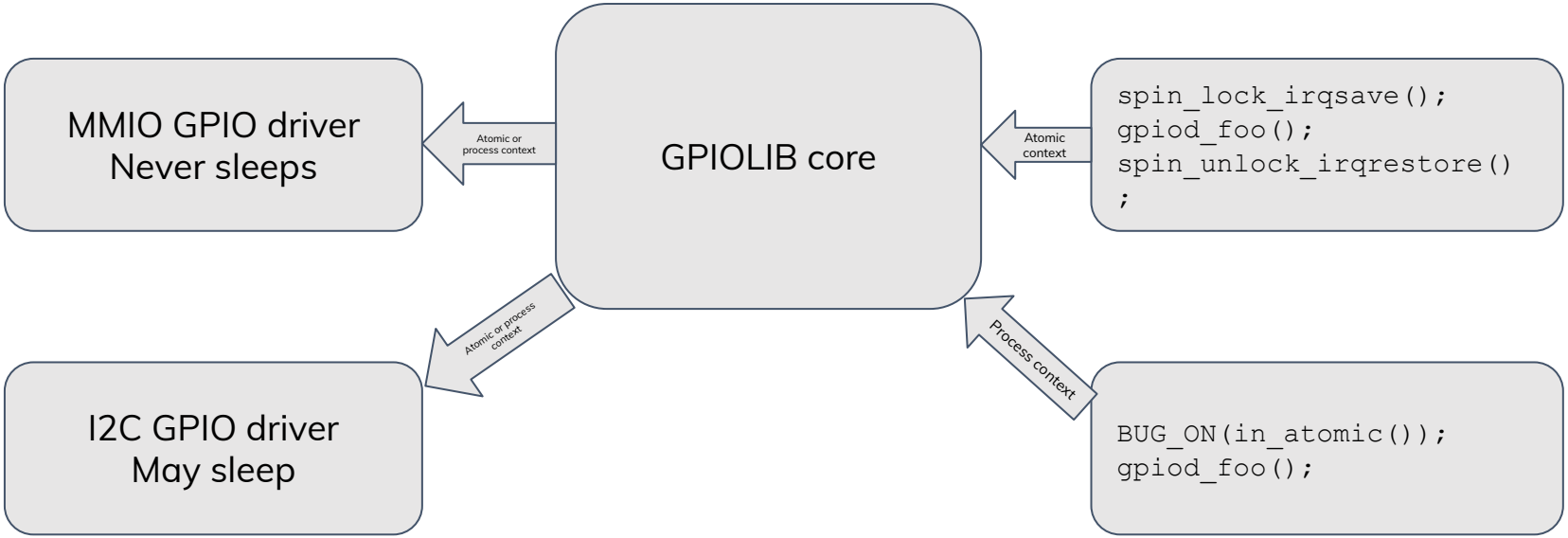


# Serialization in GPIOLIB



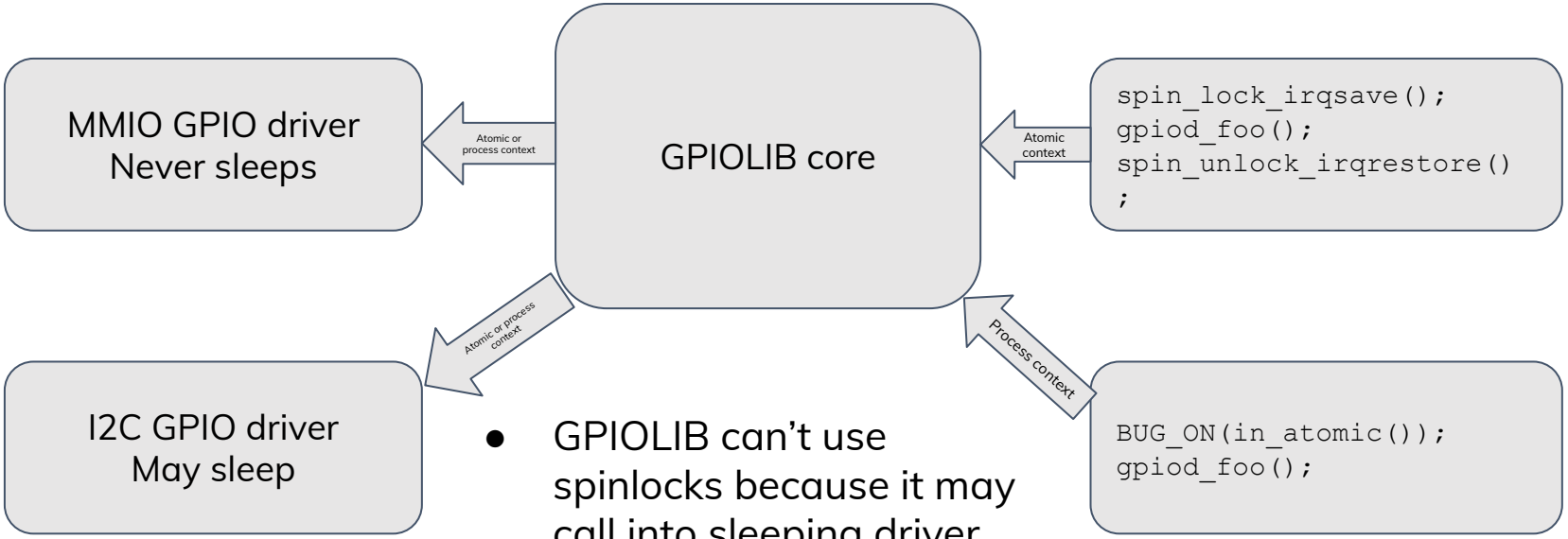

Linaro

# Serialization in GPIOLIB




Linaro

# Serialization in GPIOLIB

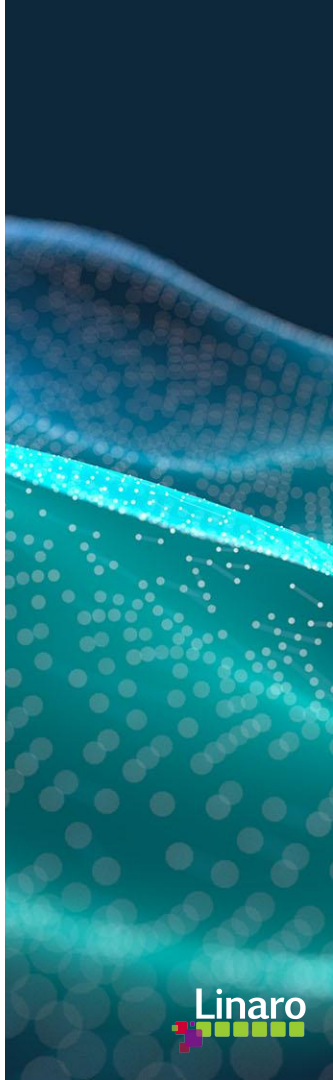



- GPIOLIB can't use spinlocks because it may call into sleeping driver callbacks
- Can't use mutexes either because it may be called from atomic context

# Serialization in GPIOLIB

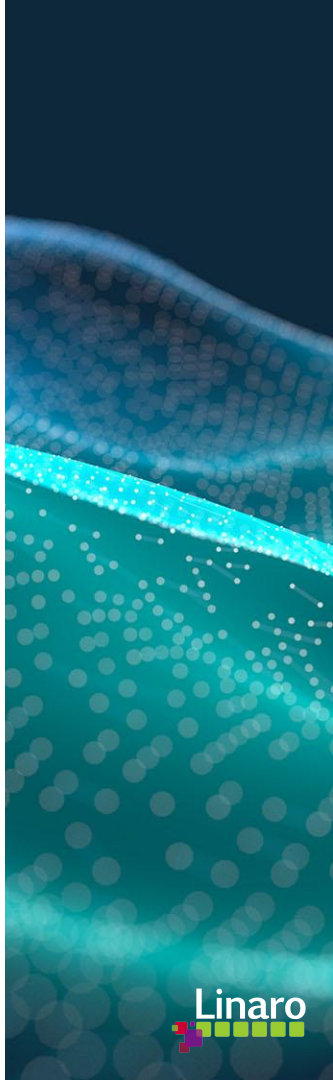
- Spinlocks would be used but badly:

```
spin_lock_irqsave();
do_atomic_stuff();
spin_unlock_irqrestore();
do_sleeping_stuff();
spin_lock_irqsave();
// Don't check state
do_atomic_stuff_again();
spin_unlock_irqrestore();
```



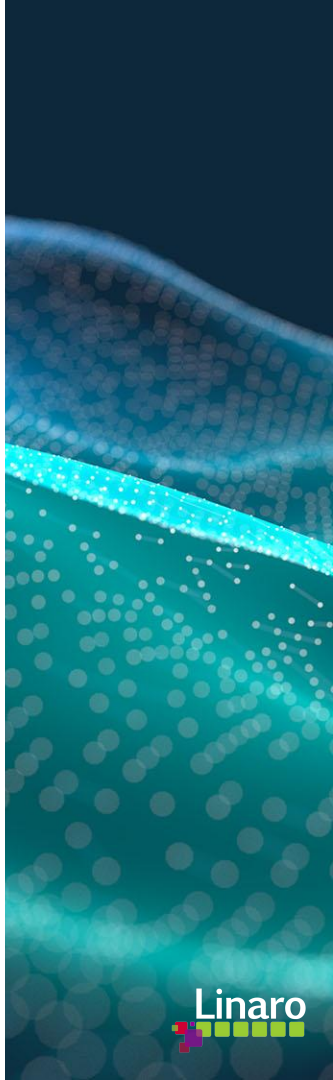
# Serialization in GPIOLIB

- If all that isn't enough: we have interrupt subsystem code calling GPIO code with spinlocks taken
- We also have pinctrl code we must call that takes mutexes

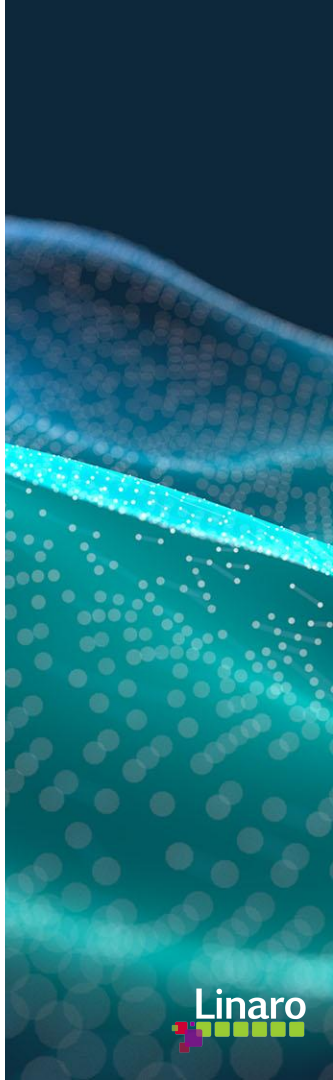


# Serialization in GPIOLIB

- A good mechanism has existed for years: SRCU
  - Read side locks can be taken from any context
  - Both sleeping and non-sleeping functions can be called with the SRCU read lock taken
- Conversion to using it took some time & effort, fixing several issues in all corners of the kernel but finally happened and is queued for linux v6.9
- GPIOLIB is now resistant to provider drivers being suddenly unbound and no longer crashes in these situations
- Still some issues to iron out with the interrupt subsystem glue code

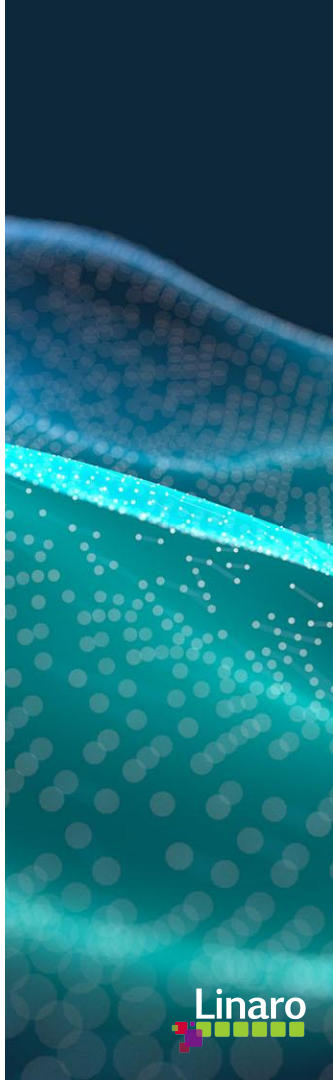


# Non-exclusive GPIOs



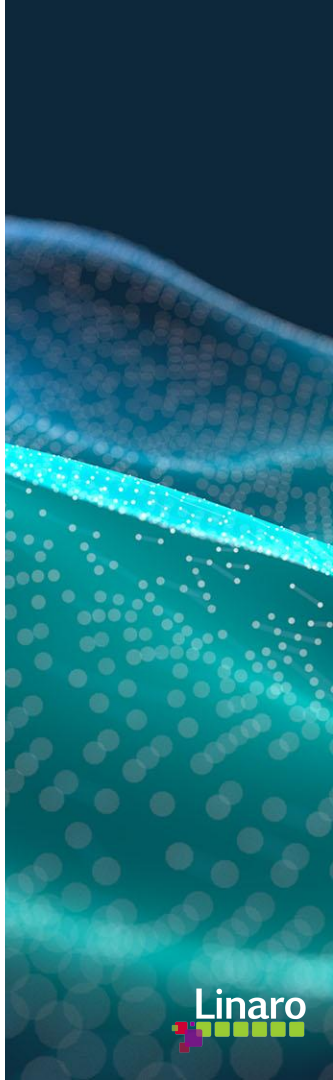
# Non-exclusive GPIOs

- Added in good faith as a workaround for fixed regulators



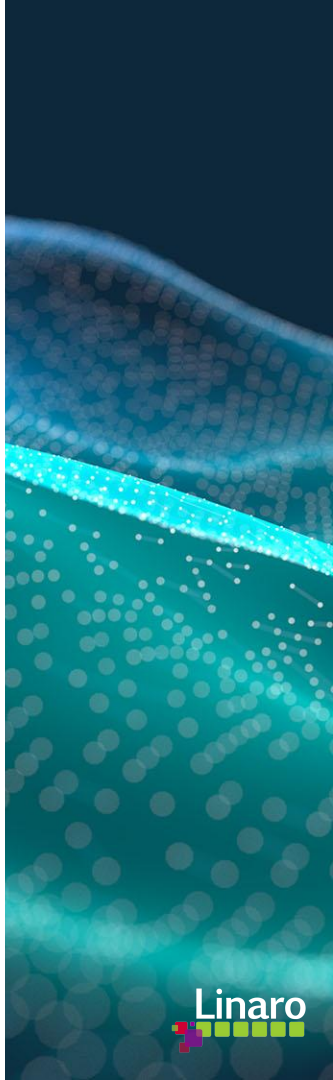
# Non-exclusive GPIOs

- Added in good faith as a workaround for fixed regulators
- Doesn’t really do what you think it does
  - Not a reference-counted GPIO enable mechanism
  - No serialization, just raw GPIO toggling, except that the descriptor is shared



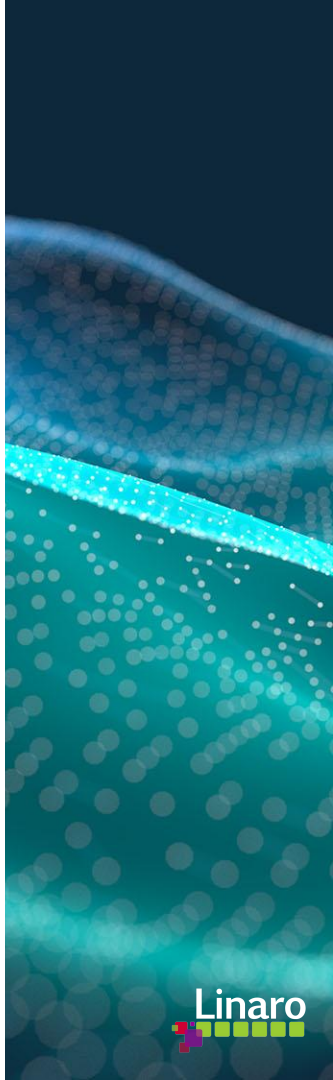
# Non-exclusive GPIOs

- Added in good faith as a workaround for fixed regulators
- Doesn’t really do what you think it does
  - Not a reference-counted GPIO enable mechanism
  - No serialization, just raw GPIO toggling, except that the descriptor is shared
- Only meant for a single use-case for regulators but others started using it as well



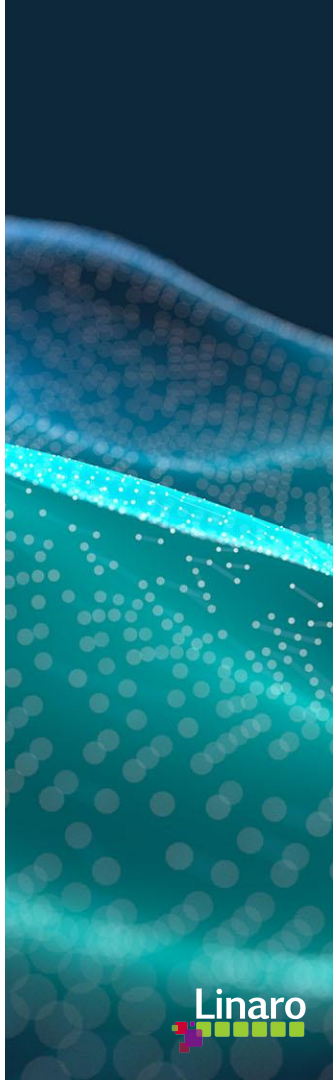
# Non-exclusive GPIOs

- Added in good faith as a workaround for fixed regulators
- Doesn’t really do what you think it does
  - Not a reference-counted GPIO enable mechanism
  - No serialization, just raw GPIO toggling, except that the descriptor is shared
- Only meant for a single use-case for regulators but others started using it as well
- We didn’t catch it and now we’re stuck until we figure out a better mechanism



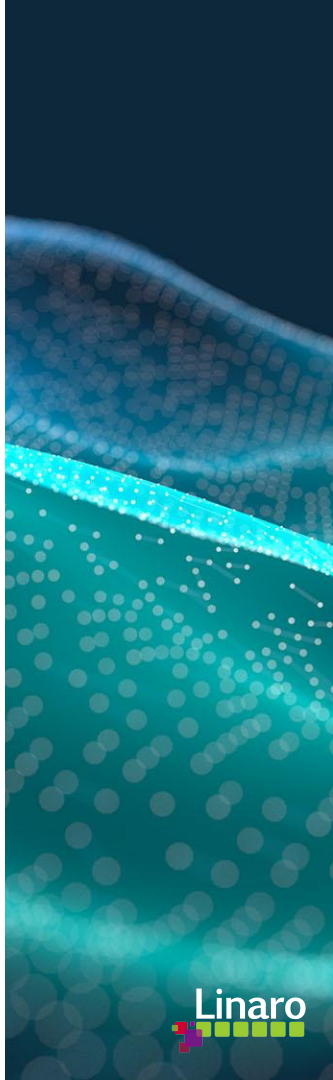
# Non-exclusive GPIOs

- Added in good faith as a workaround for fixed regulators
- Doesn’t really do what you think it does
  - Not a reference-counted GPIO enable mechanism
  - No serialization, just raw GPIO toggling, except that the descriptor is shared
- Only meant for a single use-case for regulators but others started using it as well
- We didn’t catch it and now we’re stuck until we figure out a better mechanism
- Other examples of us not catching invalid use of APIs:



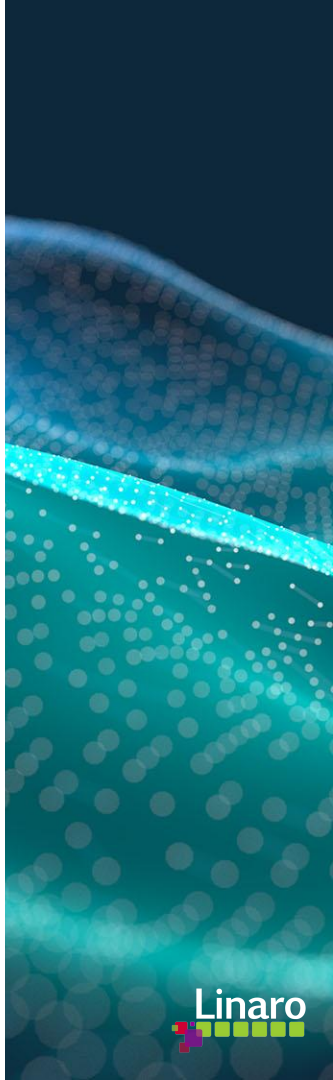
# Non-exclusive GPIOs

- Added in good faith as a workaround for fixed regulators
- Doesn’t really do what you think it does
  - Not a reference-counted GPIO enable mechanism
  - No serialization, just raw GPIO toggling, except that the descriptor is shared
- Only meant for a single use-case for regulators but others started using it as well
- We didn’t catch it and now we’re stuck until we figure out a better mechanism
- Other examples of us not catching invalid use of APIs:
  - Toggling active-low as a special workaround for MMC slots

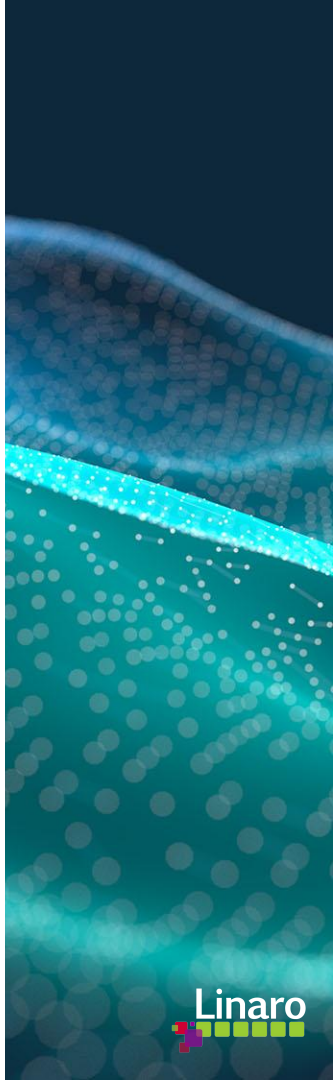


# Non-exclusive GPIOs

- Added in good faith as a workaround for fixed regulators
- Doesn’t really do what you think it does
  - Not a reference-counted GPIO enable mechanism
  - No serialization, just raw GPIO toggling, except that the descriptor is shared
- Only meant for a single use-case for regulators but others started using it as well
- We didn’t catch it and now we’re stuck until we figure out a better mechanism
- Other examples of us not catching invalid use of APIs:
  - Toggling active-low as a special workaround for MMC slots
  - Functions that were only meant to be called from process context are called while atomic

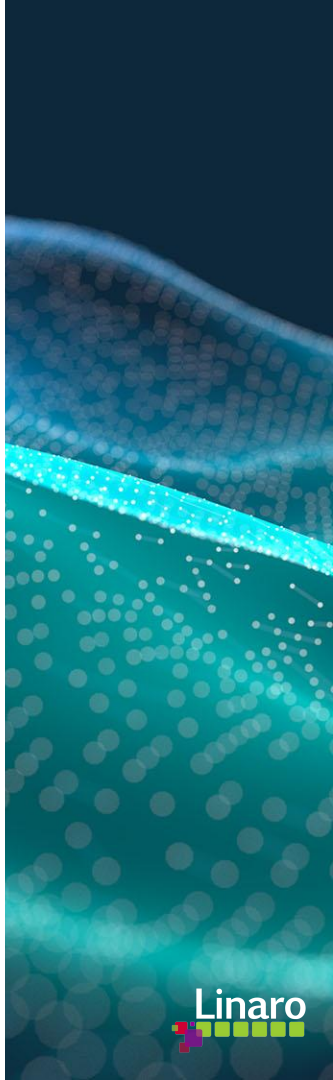


# GPIO user-space interface



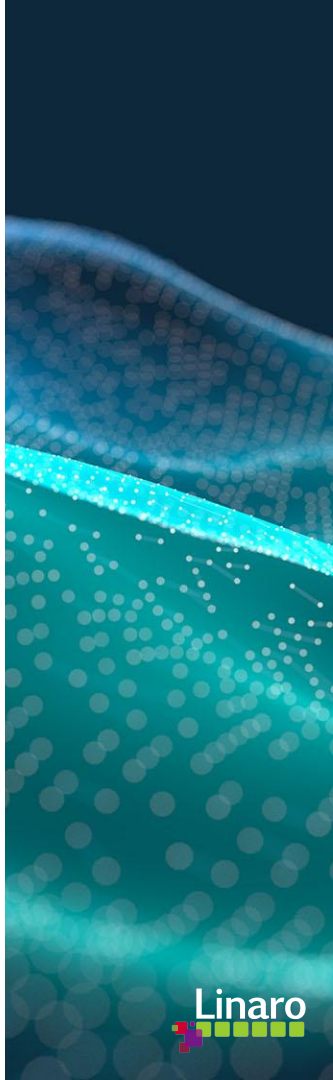
# GPIO user-space interface

- The worst kind of technical debt
  - One that cannot be addressed if user-space doesn’t willingly comply



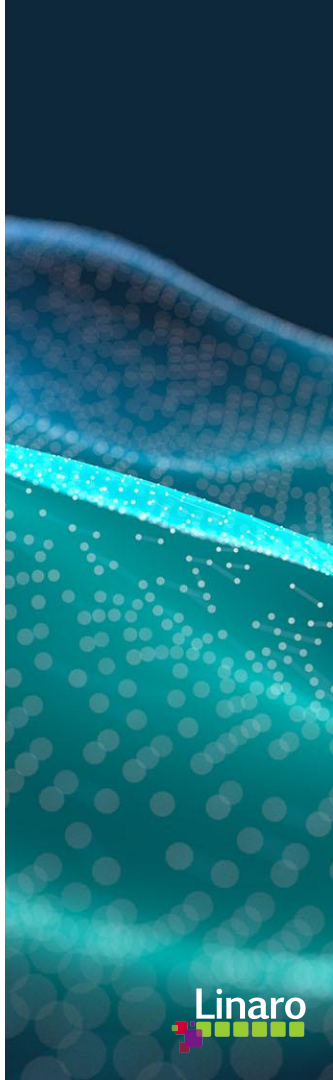
# GPIO user-space interface

- The worst kind of technical debt
  - One that cannot be addressed if user-space doesn't willingly comply
- sysfs inherits the issues of the old numberspace interface



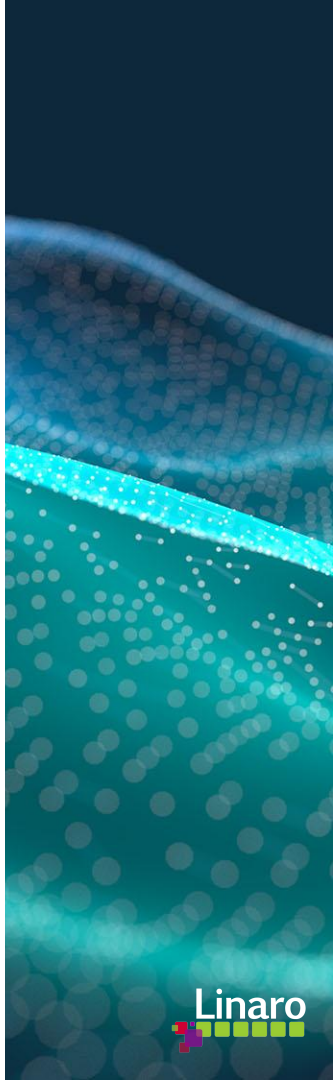
# GPIO user-space interface

- The worst kind of technical debt
  - One that cannot be addressed if user-space doesn't willingly comply
- sysfs inherits the issues of the old numberspace interface
- The character device maps the two-level, chip-line hierarchy



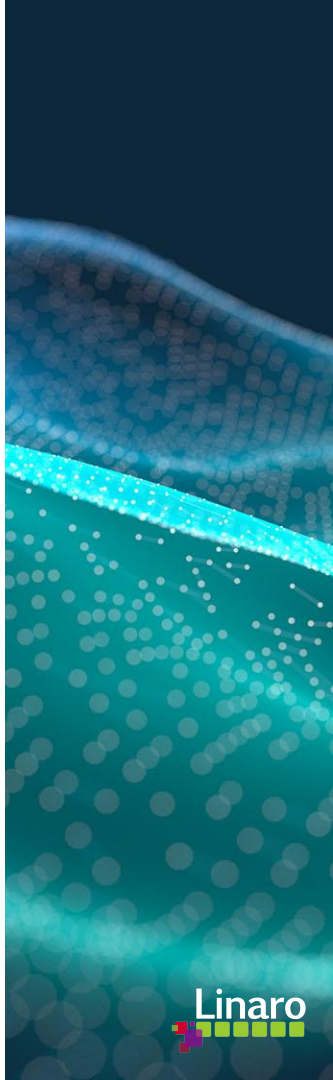
# GPIO user-space interface

- The worst kind of technical debt
  - One that cannot be addressed if user-space doesn't willingly comply
- sysfs inherits the issues of the old numberspace interface
- The character device maps the two-level, chip-line hierarchy
- Users complain about the lack of persistence of GPIO state once the character device file descriptor is closed
  - A good solution must be found to give user-space a reason to finally switch
  - DBus API is in the works



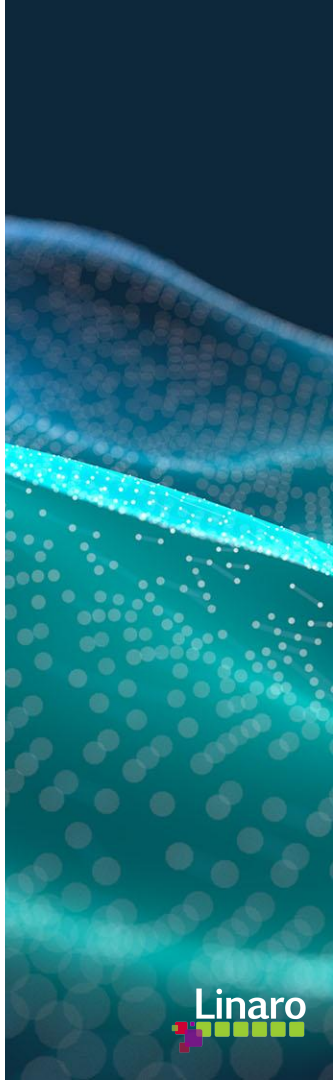
# GPIO user-space interface

- The worst kind of technical debt
  - One that cannot be addressed if user-space doesn't willingly comply
- sysfs inherits the issues of the old numberspace interface
- The character device maps the two-level, chip-line hierarchy
- Users complain about the lack of persistence of GPIO state once the character device file descriptor is closed
  - A good solution must be found to give user-space a reason to finally switch
  - DBus API is in the works
- From the kernel point of view we already have a good solution but we need more to make the user-space happy

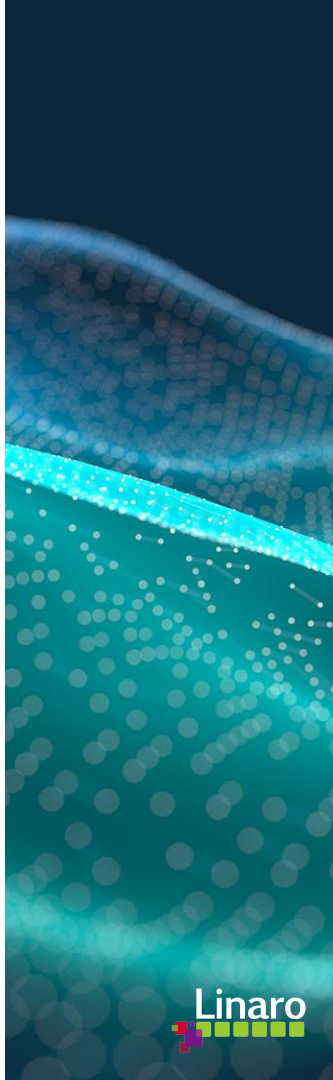


# Contain the quirks

- It’s a good idea to keep the quirk handling in a single place instead of having them scattered around

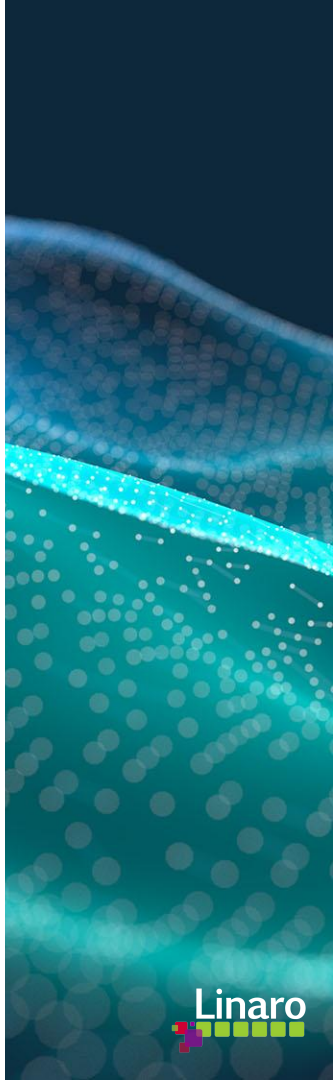


# Summary



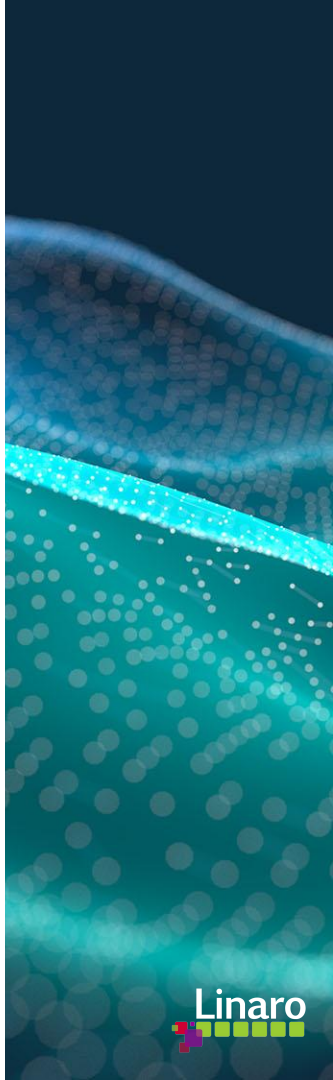
# Summary

- Old subsystems acquire cruft over time



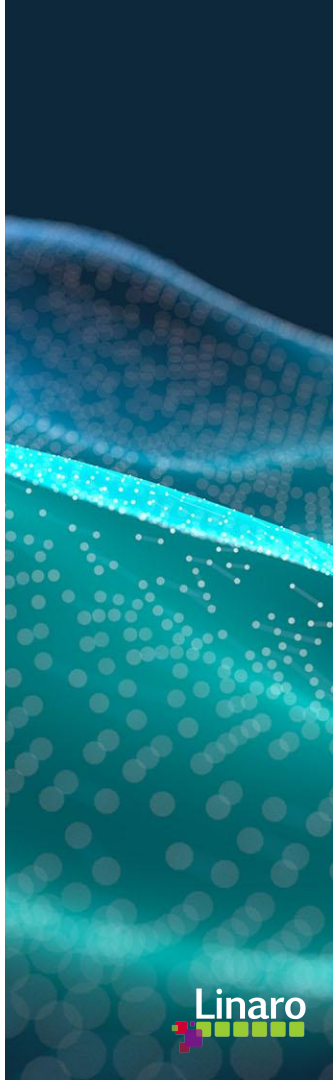
# Summary

- Old subsystems acquire cruft over time
- Commonly used subsystems are prone to having their interfaces used incorrectly
  - And it’s hard to catch it everywhere



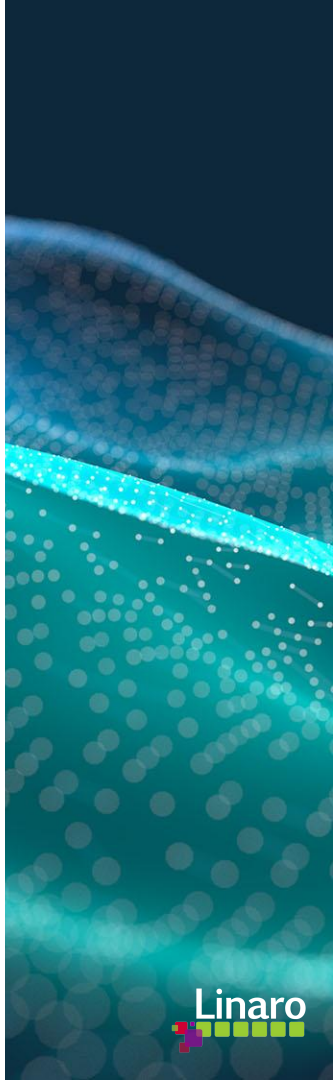
# Summary

- Old subsystems acquire cruft over time
- Commonly used subsystems are prone to having their interfaces used incorrectly
  - And it's hard to catch it everywhere
- It pays to be grumpy
  - Because the best way to solve technical debt is to never allow it in the first place



# Summary

- Old subsystems acquire cruft over time
- Commonly used subsystems are prone to having their interfaces used incorrectly
  - And it's hard to catch it everywhere
- It pays to be grumpy
  - Because the best way to solve technical debt is to never allow it in the first place
- We could use some kind of a deprecation mechanism
  - Checkpatch?
  - Coccinelle?
  - Compiler attributes



# Thank you

## Q & A

Visit www.linaro.org
Reach out to me at bartosz.golaszewski@linaro.org